中山大學 软件工程学院
SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

《Rust语言与内存安全设计》

第13讲 智能指针（续）

课程负责人：陈文清 助理教授
chenwq95@mail.sysu.edu.cn

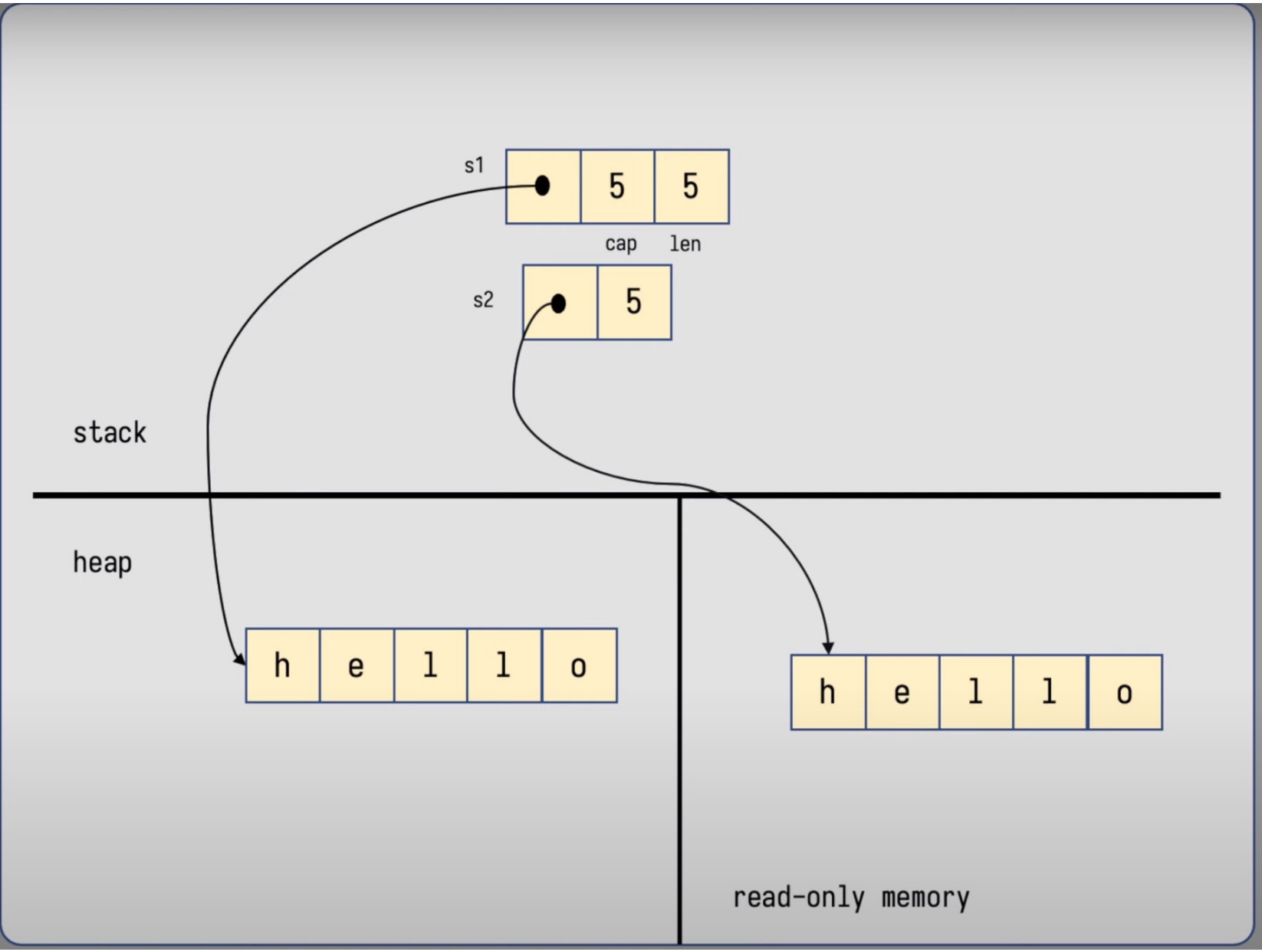
2023年11月22日

Rust中的指针类型

- 引用——安全的指针
- 原始指针
- 智能指针
  - Box<T>
  - Rc<T>
  - Arc<T>
  - Cell<T>
  - RefCell<T>

Rust中的指针类型

- **引用（Reference）——安全的指针**
  - 类似C语言中的指针，通过“&”或“&mut”符号；



String

str

&str

```
let s1: String = String::from("hello");
```

```
let s2: &str  = "hello";
```

- **String和 &str（字符串切片）**

Rust中的指针类型

■ 原始指针（Raw Pointers)

- ➢ 与引用一样，原始指针可以是不可变的或可变的，分别写为 *const T 和 *mut T;*
- ➢ *星号(*)不是解引用运算符*；它是类型名的一部分
- ➢ Unsafe Rust

- 允许同时拥有mutable和immutable借用
- 并不能保证能指向valid memory
- 允许空指针
- 没有实现自动的内存清理；
- (关于unsafe rust之后会详细展开)

```
fn main() {
    let mut num = 5;
    let r1 = &num as *const i32;
    let r2 = &mut num as *mut i32;
    unsafe{
        *r2 = *r2 + 1;
    }
    println!("{:?}", r1);
    println!("{:?}", r2);
    println!("{:?}", num);
}
```

Rust中的指针类型

■ 智能指针（Smart Pointers）

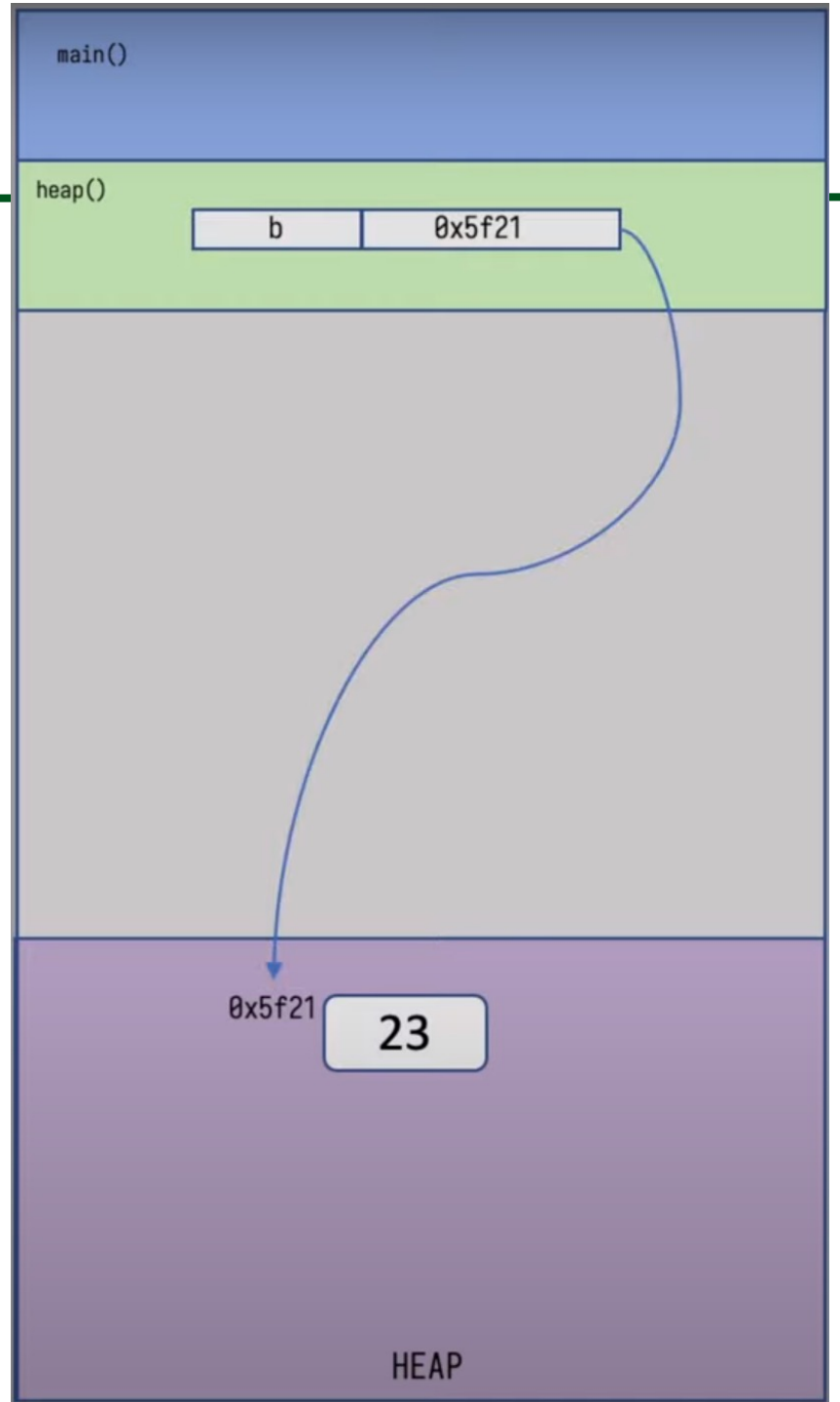
- 智能指针是一种类似于指针的数据结构，同时具有附加的元数据和功能。
- 智能指针通常使用结构体实现。与普通结构体不同，智能指针实现了 Deref 和 Drop 特性。

Rust中的指针类型

- **智能指针（Smart Pointers）**
  - Drop Trait允许你自定义在智能指针实例超出作用域时运行的代码。（执行 drop 方法）
  - Deref Trait允许智能指针结构体的实例表现得像一个引用，这样你可以编写代码以处理引用或智能指针。（解引用）

Rust中的指针类型

- **智能指针（Smart Pointers)**
  - Box<T>： Boxes allow you to store data on the heap rather than the stack.
  - 例：创建一个递归链表




```
fn main() {
    let result = heap();
}

fn heap() -> Box<i32> {
    let b = Box::new(23);
    b
}
```

- **回顾关于heap和stack上为Box<T>分配内存的案例**

Box<T>

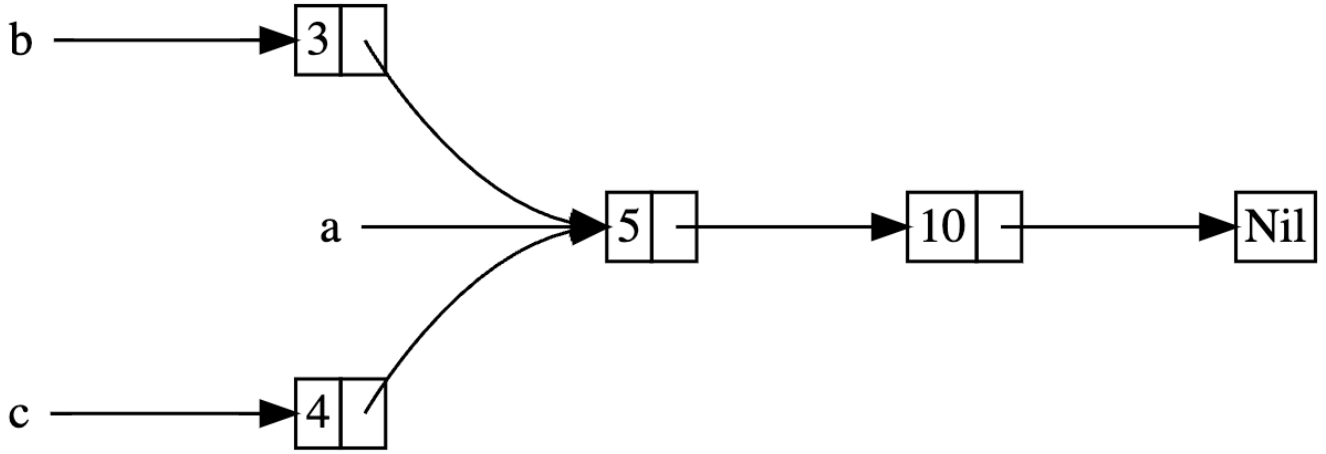
- 具有指向堆内存块的唯一指针
- Box<T>有哪些局限性？

Rc<T>

- 如果我想拥有指向同一堆内存块的多个指针，该怎么办？
- 回想一下借用规则：可以有多个不可变引用，或者最多有一个可变引用。
- Rc <T>允许你对堆内存块拥有多个不可变引用（即我们不能修改这块内存）
  - 我们为什么需要这个？
  - 答：Rust 的借用检查规则！
- 注意：如果你创建了引用循环，可能会导致内存泄漏！（如果你需要引用循环，你需要将其他智能指针类型加入到组合中）

Example: Adding Multiple Views to Our List

- 如果我们希望我们的链表能够相互“相交”，以便它们可以在数据结构不可变的情况下共享某些部分，该怎么办？（这是函数数据结构中常见的范式）
- 这可以让我们看到数据结构的“历史”！
- 这些有时被称为持久的数据结构；
- Playground示例
  - 开始
  - 结束



图片：https://doc.rust-lang.org/book/ch15-04-rc.html

Rust中的指针类型

- **智能指针（Smart Pointers)**
  - Rc<T>： Reference Counted Smart Pointer（引用计数智能指针，用于单线程）。
  - Arc<T>： Atomic Reference Counting Smart Pointer（原子引用计数智能指针，用于多线程，在之后多线程课程中介绍）。

Rust中的指针类型

■ 智能指针（Smart Pointers)

- Rc<T>： Reference Counted Smart Pointer（引用计数智能指针，用于单线程）。
  - 通过使用 Rust 类型 Rc<T> 来明确启用多所有权；
  - 想象 Rc<T> 就像家庭房间里的一台电视。
  - 当有人进来看电视时，他们打开它。
  - 其他人可以进入房间并观看电视。
  - 当最后一个人离开房间时，他们关闭电视，因为它不再被使用。

Rust中的指针类型

■ 智能指针（Smart Pointers）

➢ Rc<T>

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use crate::List::{Cons, Nil};

fn main() {
    let a = Cons(5, Box::new(Cons(10,
Box::new(Nil))));
    let b = Cons(3, Box::new(a));
    let c = Cons(4, Box::new(a));
}
```

```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use std::rc::Rc;
use crate::List::{Cons, Nil};

fn main() {
    let a = Rc::new(Cons(5,
Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}
```

Rust中的指针类型

■ 智能指针（Smart Pointers）

➢ Rc<T>

```
fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    println!("count after creating a = {}", Rc::strong_count(&a));
    let b = Cons(3, Rc::clone(&a));
    println!("count after creating b = {}", Rc::strong_count(&a));
    {
        let c = Cons(4, Rc::clone(&a));
        println!("count after creating c = {}", Rc::strong_count(&a));
    }
    println!("count after c goes out of scope = {}", Rc::strong_count(&a));
}
```

```
count after creating a = 1
count after creating b = 2
count after creating c = 3
count after c goes out of scope = 2
```

注意：上述引用为 immutable references.

help: trait `DerefMut` is required to modify through a dereference, but it is not implemented for `Rc<T>`

Rust中的指针类型

- 课堂练习

Rust中的指针类型

■ 智能指针（Smart Pointers)

- Cell<T>与RefCell<T>
  - 借用规则存在很多限制，比如不能同时存在2个mutable borrow；
  - 但当与引用计数结合时，其限制过于严格；
  - 因此Rust提供“内部可变性”，通过Cell<T>与RefCell<T>实现；
  - Neither Cell<T> nor RefCell<T> are thread safe

```
cannot assign to data in an `Rc`
trait `DerefMut` is required to modify through a
dereference, but it is not implemented for
`Rc<List>` rustc(E0594)
```

Rust中的指针类型

■ 智能指针（Smart Pointers)

- Cell<T>
  - In this example, you can see that Cell<T> enables mutation inside an immutable struct （可以看到Cell<T>允许在一个不可变的结构体内开启内部可变性）.

8.

```rust
use std::cell::Cell;

struct SomeStruct {
    regular_field: u8,
    special_field: Cell<u8>,
}

fn main(){
    let my_struct = SomeStruct {
        regular_field: 0,
        special_field: Cell::new(1),
    };
    let new_value = 100;
    // ERROR: `my_struct` is immutable
    // my_struct.regular_field = new_value;

    // WORKS: although `my_struct` is immutable, `special_field` is a `Cell`,
    // which can always be mutated
    my_struct.special_field.set(new_value);
    assert_eq!(my_struct.special_field.get(), new_value);
}
```

https://doc.rust-lang.org/std/cell/struct.Cell.html

8.

```rust
use std::cell::Cell;

struct SomeStruct {
    regular_field: u8,
    special_field: Cell<u8>,
}

fn main(){
    let my_struct = SomeStruct {
        regular_field: 0,
        special_field: Cell::new(1),
    };
    let new_value = 100;
    // ERROR: `my_struct` is immutable
    // my_struct.regular_field = new_value;

    // WORKS: although `my_struct` is immutable, `special_field` is a `Cell`,
    // which can always be mutated
    my_struct.special_field.set(new_value);
    assert_eq!(my_struct.special_field.get(), new_value);
}
```

https://doc.rust-lang.org/std/cell/struct.Cell.html

请注意Cell是拥有T类型变量的所有权（测试String）

Rust中的指针类型

■ 智能指针（Smart Pointers）

- RefCell<T>
  - A mutable memory location with dynamically checked borrow rules.
  - Cell<T> implements interior mutability by moving values in and out of the Cell<T>. To use references instead of values, one must use the RefCell<T> type.
  - Many shared smart pointer types, including Rc<T> and Arc<T>, can only be borrowed with &, not &mut. Without cells it would be impossible to mutate data inside of these smart pointers at all.

https://doc.rust-lang.org/std/cell/index.html

Rust中的指针类型

■ 智能指针（Smart Pointers）

- RefCell<T>
  - 具有“动态检查的借用规则的”可变内存位置。
  - Cell<T>通过在Cell<T>中移动值来实现内部可变性。要使用引用而不是值，必须使用RefCell<T>类型。
  - 许多共享的智能指针类型，包括Rc<T>和Arc<T>，只能通过&而不是&mut进行借用。如果没有Cell，在这些智能指针内部是无法修改数据的。

https://doc.rust-lang.org/std/cell/index.html

RefCell<T>

- RefCell 通过提供内部可变性，让你能够“欺骗”编译器
- 也就是说，你可以共享对 Cell 的引用，但你可以改变它里面的内容！

- 它的new函数不进行堆分配
- 这仍然是安全的，因为它将在“运行时”强制执行借用规则（但这现在是一个额外的成本）
- (try_)borrow/borrow_mut

借用规则：
1. 在任意给定时刻，只能拥有一个可变引用或任意数量的不可变引用（而不是两者）。
2. 引用必须总是有效的。

RefCell<T>

- RefCell 通过提供内部可变性，让你能够“欺骗”编译器
- 也就是说，你可以共享对 Cell 的引用，但你可以改变它里面的内容！

- 它的new函数不进行堆分配
- 这仍然是安全的，因为它将在“运行时”强制执行借用规则（但这现在是一个额外的成本）
- (try_)borrow/borrow_mut

- 常见模式：Rc<RefCell<T>>
- 在更复杂的数据结构中，你会经常看到这种情况，其中多个指针指向同一块数据，这需要支持可变性

Rust中的指针类型

■ 智能指针（Smart Pointers）

借用规则的一个推论是当有一个不可变值时，不能可变地借用它。例如，如下代码不能编译：

```
fn main() {
    let x = 5;
    let y = &mut x;
}
```

如果尝试编译，会得到如下错误：

```
error[E0596]: cannot borrow immutable local variable `x` as mutable
 --> src/main.rs:3:18
  |
2 |     let x = 5;
  |         - consider changing this to `mut x`
3 |     let y = &mut x;
  |                  ^ cannot borrow mutably
```

Rust中的指针类型

■ 智能指针（Smart Pointers)

借用规则的一个推论是当有一个不可变值时，不能可变地借用它。例如，如下代码不能编译：

```
fn main() {
    let x = 5;
    let y = &mut x;
}
```

如果尝试编译，会得到如下错误：

```
error[E0596]: cannot borrow immutable local variable `x` as mutable
 --> src/main.rs:3:18
  |
2 |     let x = 5;
  |         - consider changing this to `mut x`
3 |     let y = &mut x;
  |                  ^ cannot borrow mutably
```

然而，特定情况下，令一个值在其方法内部能够修改自身，而在其他代码中仍视为不可变，是很有用的。值方法外部的代码就不能修改其值了。RefCell<T> 是一个获得内部可变性的方法。

Rust中的指针类型

- 智能指针（Smart Pointers）

```
▶ Run | Debug
fn main(){
    let mut x: i32 = 5;
    let y: RefCell<i32> = RefCell::new(10);

    let a: &mut i32 = &mut x;
    let mut b: RefMut<'_, i32> = y.borrow_mut();

    *a += 1;
    *b += 1;

    println!("{}, {}", x, *b);
}
```

Rust中的指针类型

■ 智能指针（Smart Pointers)

- RefCell 在 “运行时”强制执行借用规则（但这现在是一个额外的成本）
- (try_)borrow/borrow_mut

借用规则：
1. 在任意给定时刻，只能拥有一个可变引用或任意数量的不可变引用（而不是两者）。
2. 引用必须总是有效的。

Rust中的指针类型

- 智能指针（Smart Pointers）

```
▶ Run | Debug
fn main(){
    let mut x: i32 = 5;
    let y: RefCell<i32> = RefCell::new(10);

    let a: &mut i32 = &mut x;
    let mut b: RefMut<'_, i32> = y.borrow_mut();
    let c: Ref<'_, i32> = y.borrow();

    *a += 1;
    *b += 1;

    println!("{}, {}", *c, *b);
}
```

Q:上述代码预期的运行结果？

Rust中的指针类型

■ 智能指针（Smart Pointers）

```
▶ Run | Debug
fn main(){
    let mut x: i32 = 5;
    let y: RefCell<i32> = RefCell::new(10);

    let a: &mut i32 = &mut x;
    let mut b: RefMut<'_, i32> = y.borrow_mut();
    let c: Ref<'_, i32> = y.borrow();

    *a += 1;
    *b += 1;

    println!("{}, {}", *c, *b);
}
```

```
warning: `hello_cargo` (bin "hello_cargo") generated 2 warnings
    Finished dev [unoptimized + debuginfo] target(s) in 0.13s
     Running `target/debug/hello_cargo`
thread 'main' panicked at 'already mutably borrowed: BorrowError', src/main.rs:15:15
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
(base) wenqingchen-pro:hello_cargo $
```

运行时错误而不是编译期间的错误！

Rust中的指针类型

- 智能指针（Smart Pointers）

```rust
struct SomeStruct {
    regular_field: u8,
    special_field: RefCell<u8>,
}

▶ Run | Debug
fn main(){
    let my_struct: SomeStruct = SomeStruct {
        regular_field: 0,
        special_field: RefCell::new(1),
    };
    let new_value: u8 = 100;

    let mut special_field_ref: RefMut<'_, u8> = my_struct.special_field.borrow_mut();
    *special_field_ref = new_value;
    println!("{:?}", *special_field_ref);
}
```

Rust中的指针类型

■ 智能指针（Smart Pointers）

- RefCell 在 “运行时”强制执行借用规则（但这现在是一个额外的成本）
- (try_)borrow/borrow_mut

借用规则：
1. 在任意给定时刻，只能拥有一个可变引用或任意数量的不可变引用（而不是两者）。
2. 引用必须总是有效的。

Rc<RefCell<T>>：多所有权 + 内部可变性
是否会与借用规则冲突呢？

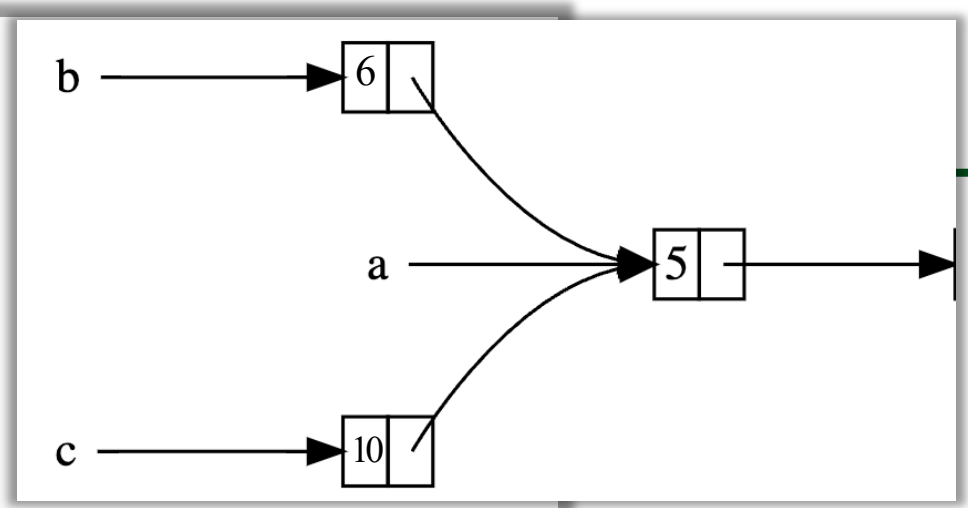
Rust中的指

- 智能指针（S

```
#[derive(Debug)]
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::rc::Rc;
use std::cell::RefCell;

fn main() {
    let value = Rc::new(RefCell::new(5));

    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));

    let b = Cons(Rc::new(RefCell::new(6)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(10)), Rc::clone(&a));

    *value.borrow_mut() += 10;

    println!("a after = {:?}", a);
    println!("b after = {:?}", b);
    println!("c after = {:?}", c);
}
```

```
a after = Cons(RefCell { value: 15 }, Nil)
b after = Cons(RefCell { value: 6 }, Cons(RefCell { value: 15 }, Nil))
c after = Cons(RefCell { value: 10 }, Cons(RefCell { value: 15 }, Nil))
```

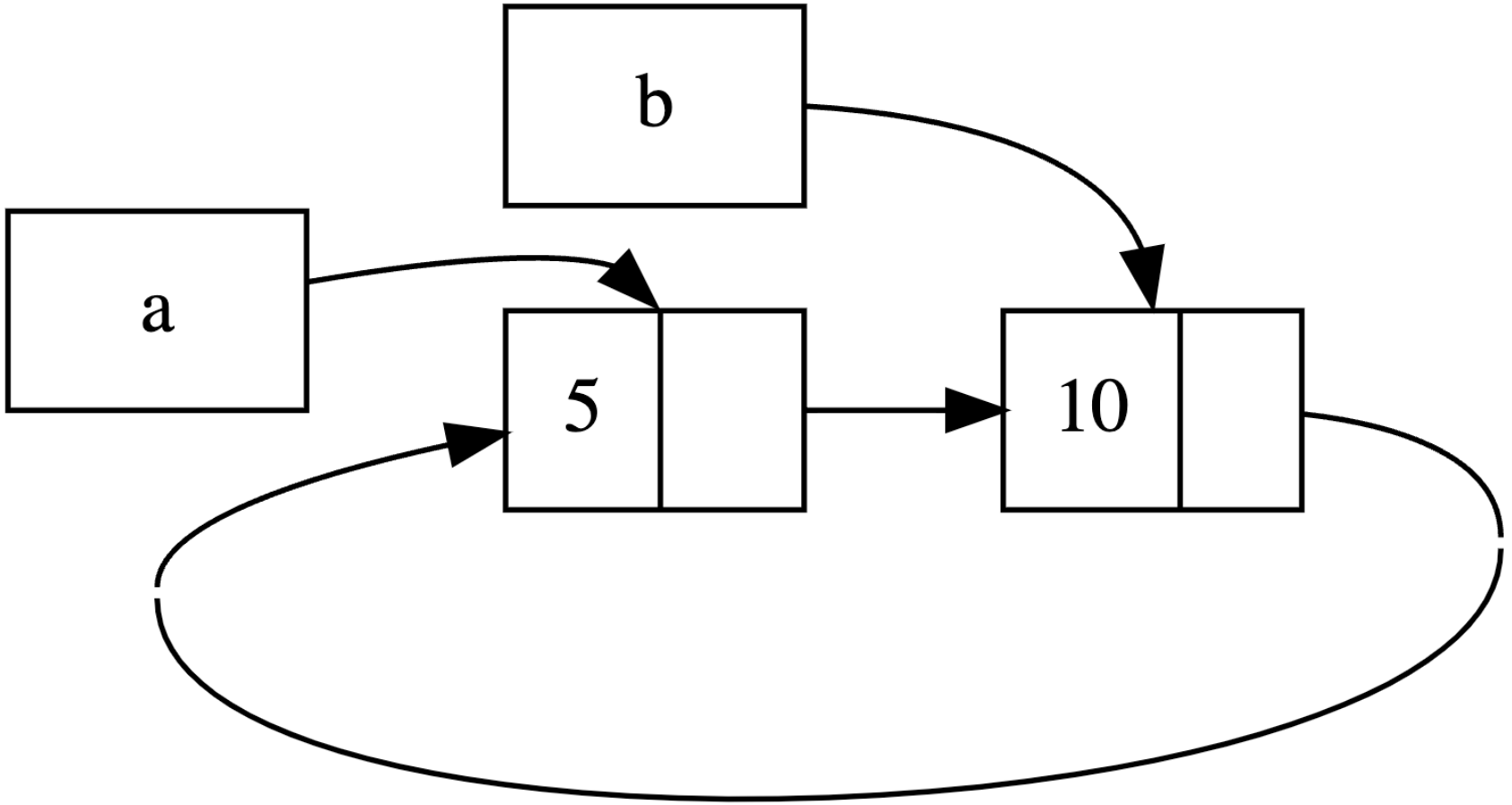
```rust
#[derive(Debug)]
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::rc::Rc;
use std::cell::RefCell;

fn main() {
    let value = Rc::new(RefCell::new(5));

    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));

    let b = Cons(Rc::new(RefCell::new(6)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(10)), Rc::clone(&a));

    *value.borrow_mut() += 10;

    println!("a after = {:?}", a);
    println!("b after = {:?}", b);
    println!("c after = {:?}", c);
}
```




```
a after = Cons(RefCell { value: 15 }, Nil)
b after = Cons(RefCell { value: 6 }, Cons(RefCell { value: 15 }, Nil))
c after = Cons(RefCell { value: 10 }, Cons(RefCell { value: 15 }, Nil))
```

引用循环与内存泄漏

- 智能指针（Smart Pointers）



引用循环

- 智能指针

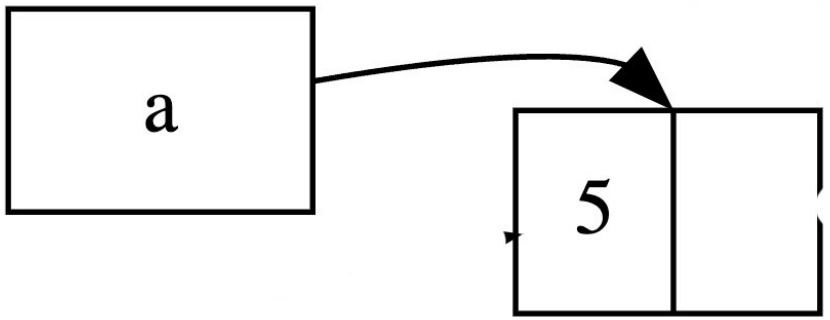
```
use std::rc::Rc;
use std::cell::RefCell;
use crate::List::{Cons, Nil};

#[derive(Debug)]
enum List {
    Cons(i32, RefCell<Rc<List>>),
    Nil,
}

impl List {
    fn tail(&self) -> Option<&RefCell<Rc<List>>> {
        match self {
            Cons(_, item) => Some(item),
            Nil => None,
        }
    }
}
```

```
fn main() {
    let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));

    println!("a initial rc count = {}", Rc::strong_count(&a));
    println!("a next item = {:?}", a.tail());

    let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));

    println!("a rc count after b creation = {}", Rc::strong_count(&a));
    println!("b initial rc count = {}", Rc::strong_count(&b));
    println!("b next item = {:?}", b.tail());

    if let Some(link) = a.tail() {
        *link.borrow_mut() = Rc::clone(&b);
    }

    println!("b rc count after changing a = {}", Rc::strong_count(&b));
    println!("a rc count after changing a = {}", Rc::strong_count(&a));

    // Uncomment the next line to see that we have a cycle;
    // it will overflow the stack
    // println!("a next item = {:?}", a.tail());
}
```

初始化a

引用循环与内存泄漏

- 智能指针（Smart Pointers）

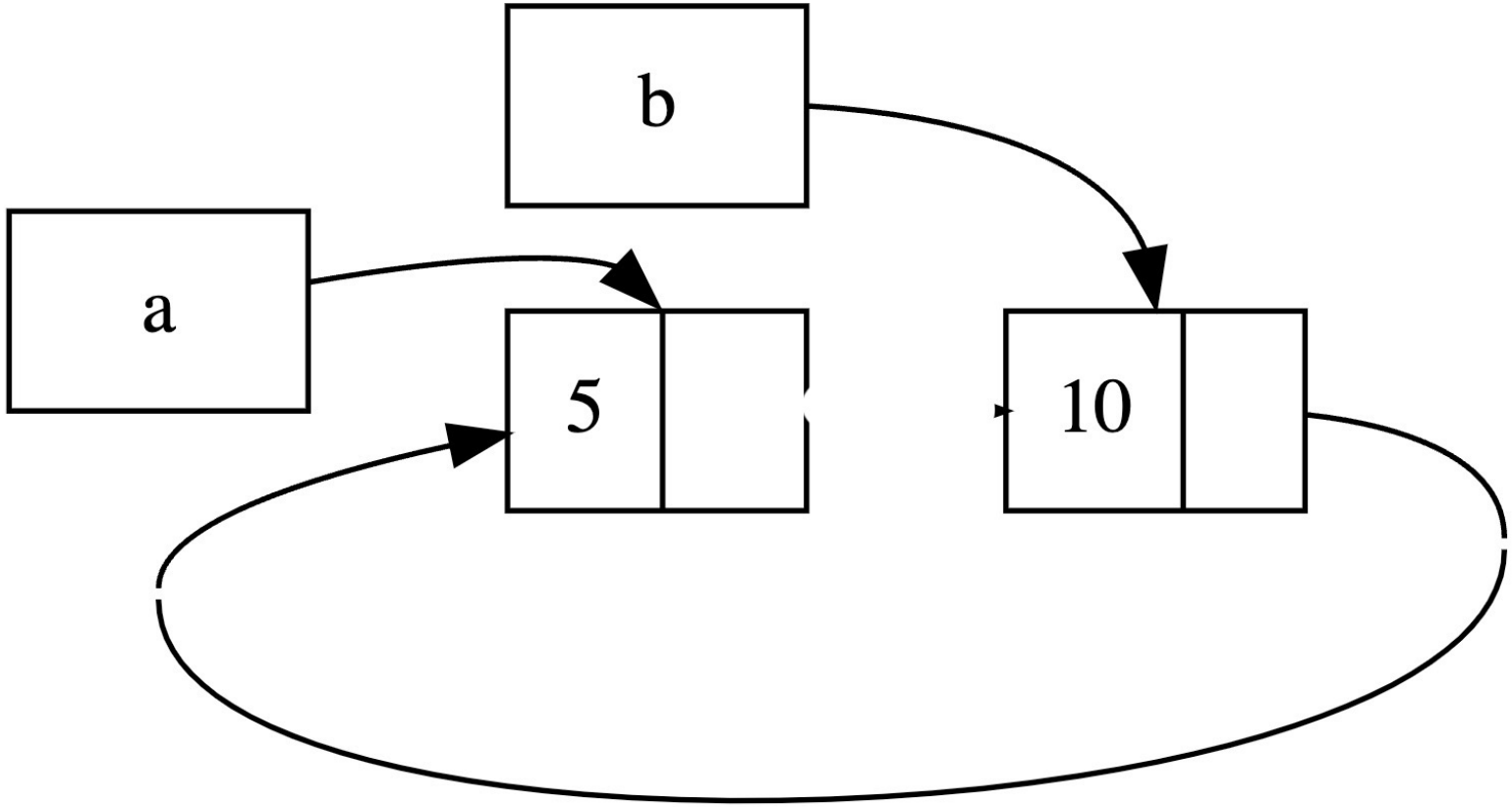


```
fn main() {
    let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));

    println!("a initial rc count = {}", Rc::strong_count(&a));
    println!("a next item = {:?}", a.tail());

    let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));

    println!("a rc count after b creation = {}", Rc::strong_count(&a));
    println!("b initial rc count = {}", Rc::strong_count(&b));
    println!("b next item = {:?}", b.tail());

    if let Some(link) = a.tail() {
        *link.borrow_mut() = Rc::clone(&b);
    }

    println!("b rc count after changing a = {}", Rc::strong_count(&b));
    println!("a rc count after changing a = {}", Rc::strong_count(&a));

    // Uncomment the next line to see that we have a cycle;
    // it will overflow the stack
    // println!("a next item = {:?}", a.tail());
}
```

这一步执行完，b指向了a
如下页PPT所示

引用循环与内存泄漏

- 智能指针（Smart Pointers）

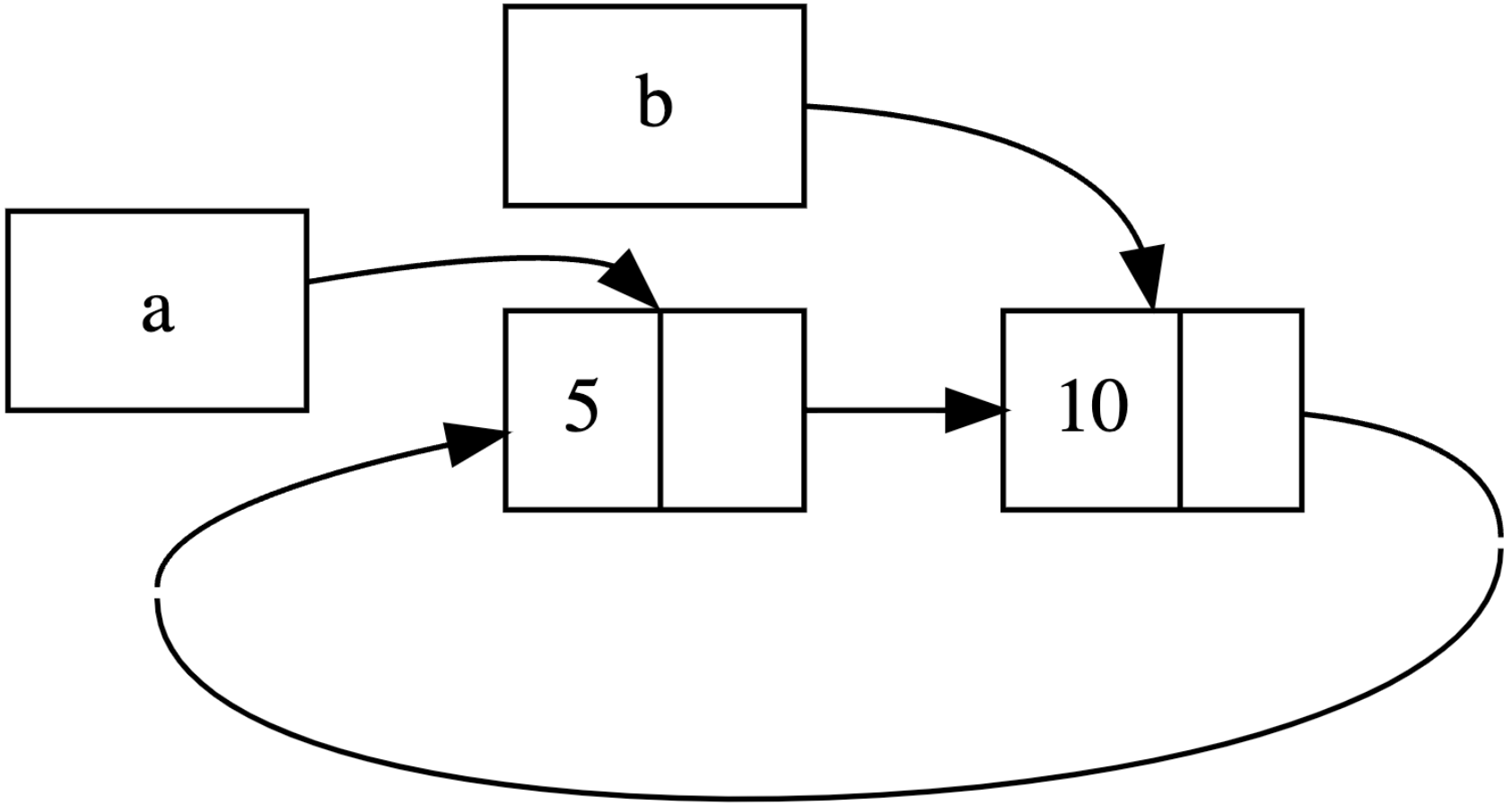


```
fn main() {
    let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));

    println!("a initial rc count = {}", Rc::strong_count(&a));
    println!("a next item = {:?}", a.tail());

    let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));

    println!("a rc count after b creation = {}", Rc::strong_count(&a));
    println!("b initial rc count = {}", Rc::strong_count(&b));
    println!("b next item = {:?}", b.tail());

    if let Some(link) = a.tail() {
        *link.borrow_mut() = Rc::clone(&b);
    }

    println!("b rc count after changing a = {}", Rc::strong_count(&b));
    println!("a rc count after changing a = {}", Rc::strong_count(&a));

    // Uncomment the next line to see that we have a cycle;
    // it will overflow the stack
    // println!("a next item = {:?}", a.tail());
}
```

因为a.tail()返回的是RefCell
内部可变性的用法，a指向了b

引用循环与内存泄漏

- 智能指针（Smart Pointers）



引用循环与内存泄漏

- 智能指针（Smart Pointers）

```
41      // it will overflow the stack
42      println!("a next item = {:?}", a.tail());
43  }
44
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
 value: Cons(5, RefCell { value: Cons(10, RefCell { value: Cons(5, RefCell {
Cons(10, RefCell { value: Cons(5, RefCell { value: Cons(10, RefCell { value:
 RefCell { value: Cons(10, RefCell { value: Cons(5, RefCell { value: Cons(10
l { value: Cons(5, RefCell { value: Cons(10, RefCell { value: Cons(5, RefCel
e: Cons(10, RefCell { value: Cons(5, RefCell { value: Cons(10, RefCell { val
thread 'main' has overflowed its stack
fatal runtime error: stack overflow
Abort trap: 6
(base) wengingchen-pro:hello_cargo $
```

将Rc<T>变为Weak<T>

■ 智能指针（Smart Pointers)

- Rc::downgrade 代替Rc::clone
  - 通过调用 Rc::downgrade 并传递 Rc<T> 实例的引用来创建其值的**弱引用**（*weak reference*）

- 不同于将 Rc<T> 实例的 strong_count 加1，调用 Rc::downgrade 会将 weak_count 加1

- weak_count 无需计数为 0 就能使 Rc<T> 实例被清理。弱引用并不属于所有权关系。

- 为了使用 Weak<T> 所指向的值，我们必须确保其值仍然有效。为此可以调用 Weak<T> 实例的 upgrade 方法，这会返回 Option<Rc<T>>

将Rc<T>变为Weak<T>

```rust
fn main() {
    let data: Rc<String> = Rc::new("example".to_string());

    let strong_count: usize = Rc::strong_count(this: &data);
    println!("Strong count: {}", strong_count); // 输出 1

    let weak_count: usize = Rc::weak_count(this: &data);
    println!("Weak count: {}", weak_count); // 输出 0

    let cloned_data: Rc<String> = Rc::clone(self: &data);
    let new_strong_count: usize = Rc::strong_count(this: &data);
    println!("New strong count: {}", new_strong_count); // 输出 2

    let weak_reference: Weak<String> = Rc::downgrade(this: &data);
    let new_weak_count: usize = Rc::weak_count(this: &data);
    println!("New weak count: {}", new_weak_count); // 输出 1
}
```

weak_count统计的是弱引用数量

小结

- 引用——安全的指针
- 原始指针
- 智能指针
  - Box<T>
  - Rc<T>
  - Arc<T>
  - Cell<T>
  - RefCell<T>

第三次作业

作业要求：

1.创建二叉树节点结构体：

1. 创建一个二叉树节点结构体类型TreeNode，包含一个整数值和两个子节点（见下页PPT）。

2.使用 Rc<T> 管理节点引用：

1. 使用 Rc 来允许多个节点之间共享相同的子节点。

3.使用 RefCell <T> 允许动态可变性：

1. 使用 RefCell 来允许在节点中进行动态可变操作，例如修改节点的值。

4.实现二叉树功能：

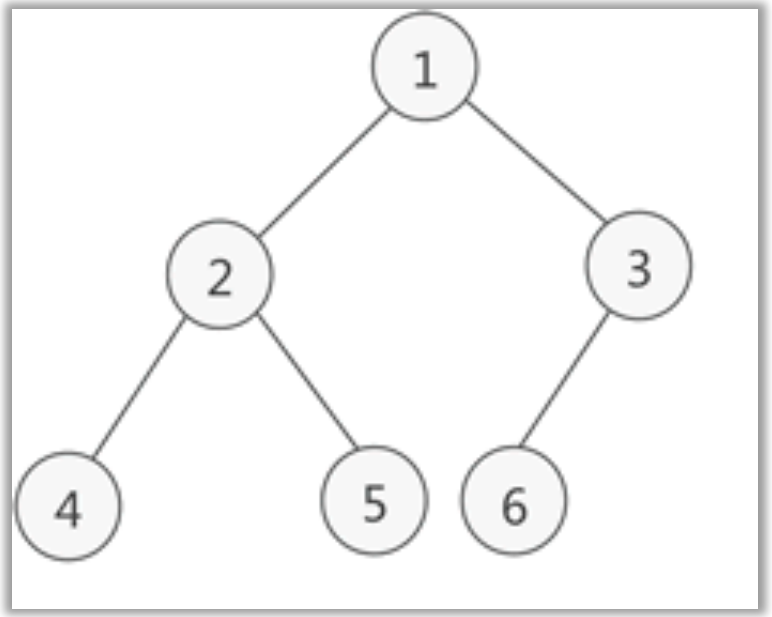
1. 在 main 函数中创建一个简单的二叉树，**包含6个节点，其形状见下页PPT**
2. 实现二叉树的中序遍历算法，并使用 println! 打印节点值。

5.编写测试用例：

1. 编写测试用例，使用 cargo test 进行测试，确保二叉树的基本功能和智能指针的正确使用。
2. 测试包括创建节点、连接节点、修改节点值、遍历二叉树等方面。

第三次作业

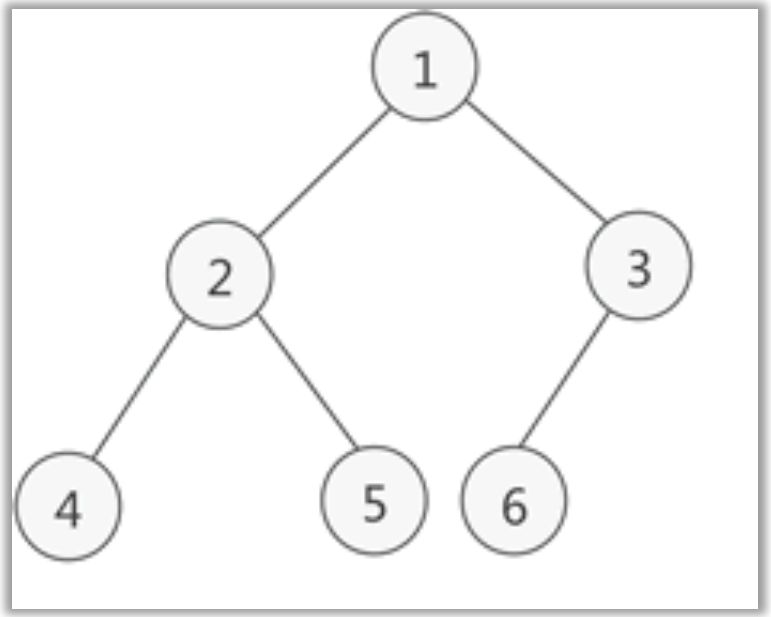
二叉树的定义如右图：




```rust
use std::rc::Rc;
use std::cell::RefCell;

#[derive(Debug)]
2 implementations
struct TreeNode {
    value: i32,
    left: Option<Rc<RefCell<TreeNode>>>,
    right: Option<Rc<RefCell<TreeNode>>>,
}

impl TreeNode {
    fn new(value: i32) -> Rc<RefCell<TreeNode>> {
        Rc::new(RefCell::new(TreeNode {
            value,
            left: None,
            right: None,
        }))
    }
}
```

第三次作业

```rust
fn main() {
    // Create a simple binary tree
    let root: Rc<RefCell<TreeNode>> = TreeNode::new(1);
    // to be done
}
```




```rust
#[cfg(test)]
▶ Run Tests | Debug
mod tests {
    use super::*;
    #[test]
    ▶ Run Test | Debug
    fn test_tree_creation() {
        // Create nodes
        // to be done
    }
    #[test]
    ▶ Run Test | Debug
    fn test_tree_modification() {
        // Create nodes
        // to be done
    }
    #[test]
    ▶ Run Test | Debug
    fn test_in_order_traversal() {
        // Create nodes
        // to be done
    }
}
```

第三次作业

2023秋rust课程第三次作业收集（为了简便起见，仅提交一个main.rs文件，把测试用例写到main.rs中）

截止时间：2023-11-29 23:59
提交地址：https://send2me.cn/20Dhnas1/SkquN551RHXxGw

Q & A

Thanks!